Education Tech Research Dev (2023) 71:2365–2388
https://doi.org/10.1007/s11423-023-10277-2

DEVELOPMENT ARTICLE

# Automated assessment system for programming courses: a case study for teaching data structures and algorithms

**Andre L. C. Barczak[1,2] · Anuradha Mathrani[2] · Binglan Han[2] · Napoleon H. Reyes[2]**

Accepted: 30 July 2023 / Published online: 15 August 2023


## Abstract

An important course in the computer science discipline is '*Data Structures and Algorithms' (DSA). The coursework* lays emphasis on experiential learning for building students' programming and algorithmic reasoning abilities. Teachers set up a repertoire of formative programming exercises to engage students with different programmatic scenarios to build their know-what, know-how and know-why competencies. Automated assessment tools can assist teachers in inspecting, marking, and grading of programming exercises and also support them in providing students with formative feedback in real-time. This article describes the design of a bespoke automarker that was integrated into the DSA coursework and therefore served as an instructional tool. Activity theory has provided the pedagogical lens to examine how the automarker-mediated instructional strategy enabled self-reflection and assisted students in their formative learning journey. Learner experiences gathered from 39 students enrolled in DSA course shows that the automarker facilitated practice-based learning to advance students know-what, know-why and know-how skills. This study contributes to both curricula and pedagogic practice by showcasing the integration of an automated assessment strategy with programming-related coursework to inform future teaching and assessment practice.

**Keywords** Automated assessment · Activity theory · Programming · Formative learning · Pedagogy · Practice-driven learning

✉ Andre L. C. Barczak
abarczak@bond.edu.au; a.l.barczak@massey.ac.nz

Anuradha Mathrani
A.S.Mathrani@massey.ac.nz

Binglan Han
B.Han1@massey.ac.nz

Napoleon H. Reyes
N.H.Reyes@massey.ac.nz

[1] Bond Business School, Bond University, Gold Coast, QLD 4226, Australia

[2] School of Mathematical and Computational Sciences, Massey University - Auckland Campus, Private Bag 102904, North Shore City, Auckland 0745, New Zealand

**Abbreviations**

| | |
|---|---|
| CS | Computer sciences |
| AA | Automated assessment |
| DSA | Data structures and algorithms |
| AT | Activity theory |
| VPN | Virtual private network |

## Introduction

As the computer science discipline (CS) continues to advance, educators apply innovative pedagogies that have greater focus on experiential learning to prepare students for computing careers. The key tenet underlying CS courses is that of writing effective programming code, since this helps demonstrate learner aptitudes for higher-order computational thinking, algorithmic reasoning, problem decomposition, iteration, recursion and overall, predisposition towards dealing with code complexities (Lemay et al., 2021). The computing curricula task force (2021) emphasizes that graduates from this discipline must be able to apply know-what, know-how and know-why skills that are indicative of professional practice. Know-what relates to "factual understanding" of the subject matter or "topics in the syllabi", which must be acted upon with a degree of proficiency to activate the know-how skills. Know-how demands time and practice that requires "engagement in a progressive hierarchy of higher-order cognitive process". Finally, know-why denotes the values and motivations that "moderate the behavior of applying "know-what" that becomes "know-how"' (p. 48). Therefore, by integrating meaningful lab activities into coursework, students can move from following simple step-by-step instructions (or "know-what goals") to engage in practical understanding, application and synthesis of learned knowledge (or "know-how" goals) to much higher-order thinking based on the newly acquired conceptual knowledge (lumped together as "know-why" goals) (Zvacek, 2015) However, it is generally acknowledged that teaching practical courses such as programming is particularly challenging (Daradoumis et al., 2019; Watson & Li, 2014), and there is little understanding of how novice programmers develop coding proficiencies (Luxton-Reilly et al., 2018). Writing effective code is an ever-evolving craft that is practice-driven. Programmers must have good understanding of the code syntax, program structures and logical expressions that together inform the practice of writing effective code (Parsons et al., 2016). Instructors therefore reinforce student learning by integrating a variety of programmatic scenarios that embody some key coding concept in the CS discipline. Programming exercises with varying levels of difficulty form a repertoire of formative learning activities to engage students and develop their programming capabilities. However, instructors are often constrained in providing individual feedback, which may be due to lack of time, large student enrolments, or simply due to the nature of the course delivery format in particular institutions (Medeiros et al., 2019). Moreover, manually evaluating students' code submissions is repetitive and time-intensive, that causes additional teaching workload. Furthermore, two teachers marking the same assessment may rarely apply the same criteria, so a student's mark may vary based on the teacher who assessed their code (Insa & Silva, 2018); therefore, manual marking is indeed subjective having inconsistencies that are not always fair to students.

Now, automated assessment (AA) tools have provided a dual perspective. First, they ease the mundane nature of marking and grading assignments for teachers, and second, these tools can facilitate prompt formative feedback to students (Skalka & Drlik, 2020;

Souza et al., 2016). AA tools rely on a coherent rubric thereby removing subjectivity concerns particularly those associated with variance due to teachers' marking styles or other forms of inconsistencies that may crop up with manual marking. Therefore, in meeting the realities of ongoing formative assessments, automation can bring more reliability in grading. With this in view, a huge demand exists among software houses and training programs on building AA capabilities to enhance e-assessment strategies (Ullah et al., 2018). Many web-based tools that incorporate static and dynamic analyses of the code snippets have emerged (Amelung et al., 2011; Staubitz et al., 2015; Ullah et al., 2018); however, these are often limited by strict file formatting guidelines (e.g., character-character equivalence across text files), broad error categories (e.g., "wrong answer", "presentation error", "compile-time error") or incompatible compiler versions which can confuse students (Rubio-Sánchez et al., 2012).

This study expands on the current state of AA research specific to the 'Data Structures and Algorithms' (DSA) course being taught at a tertiary institution. Drawing upon the limitations of currently available AA tools, we propose the design and development of our bespoke AA tool (henceforth referred as the automarker). Activity theory (AT) has provided the pedagogical lens in aligning the automarker with the DSA coursework (Engeström, 1999). The foundational concept in AT is how each 'activity' fits in a specified learning environment. Seven prescribed coursework activities that inform the DSA coursework underpin the students know-what, know-how and know-why skills. Finally, a survey of students has revealed their perceptions towards the use of the automarker as a pedagogical tool in DSA coursework. This study therefore contributes to the advancement in the field of e-assessments with use of specific examples of instructional activities for a given coursework in the CS discipline and brings awareness on how students can build up their knowledge competencies.

## Related works

High student–teacher ratio in CS courses puts extra demands on instructors as they have to mark student programming assignments and further provide students with effective feedback, all of which is very time-consuming. AA tools can reduce teachers' workload drastically (Gordillo, 2019), since these tools can automate these tasks by integrating marking/grading functionalities via test cases that refer to the executable code submissions. Comparisons are made between test case results specified from a model solution provided by the teacher and that of the student code, to see whether both results are identical, although some permutations may be allowed for exercises, such as those involving a list-valued output (Amelung et al., 2011). Some commonly used plugins that incorporate AA approaches include Algo+ (Belhaoues et al., 2016), EFPL (Bey et al., 2018), WebCAT (Manzoor et al., 2020) and CodeRunner (Soll et al., 2021). However, AA tools need careful consideration; teachers must pay specific attention to the pedagogical design of programming activities, since programming is driven by practice and is "not an exact science"(Bey et al., 2018, p. 260). Incorrect application of AA tools can negatively impact student engagement and performance, for instance, students may rely on such tools by way of numerous trial-and-error submissions until a correct response is received (Amelung et al., 2011; Rubio-Sánchez et al., 2012) rather than leverage them for critical thinking or building problem-solving capabilities. Hence, before implementing any AA tool, teachers must ensure that it serves the educational goals of building professional practice and encourages learner reflection.

The computing curricula (2021) is broadly scoped to cover vital areas such as computer science, information technology, information systems and software engineering; however, the overarching framework of this curriculum specifically mentions that 'the ability to develop advanced algorithms and data structures [are] developed in computer science' (p. 29). The CS curriculum prescribes DSA as a fundamental software course that lays emphasis on the discovery of programmatic approaches which can then be applied to datasets that are arranged in some specific order (e.g., lists, stacks, queues, trees, maps or graphs). DSA coursework exposes students to algorithms that make efficient use of computer resources (e.g., time and memory) to solve data problems (e.g., search, sort, group, etc.) wherein data structures hold the operational data. Understanding the dynamics of performing operations on data structures via algorithmic methods can be confusing for students (Su et al., 2021); as such DSA course offers a practice-based learning environment that uses different programmatic scenarios (Restrepo-Calle et al., 2019). Moreover, AA tools can assist teachers in evaluating the algorithmic execution steps across various lesson topics that form part of the DSA coursework (Belhaoues et al., 2016; Gárcia-Mateos & Fernández-Alemán, 2009; Soll et al., 2021).

Gárcia-Mateos and Fernández-Alemán (2009) used the AA tool (Mooshak) as an online judging strategy, in which all student marks were made public so that each student could see their own performance relative to their classmates. Those students who passed all the problem exercises did not have to do the final exam. The authors consider public ranking promotes competitiveness and motivates students to perform better. Although it can also be speculated that such ranking experiences could discourage students if they are not performing as well as their classmates. A drawback of Mooshak was that incorrect submissions were broadly classified as "wrong answer"; hence, the tool could not be leveraged by students in fully developing their know-what, know-how or know-why skills. Rubio-Sánchez et al. (2012) too found poor acceptance for Mooshak in their study. The authors point out that broad feedback responses returned from Mooshak could be for many reasons. These could be if Mooshak's compiler version is different to the student's version resulting in a "wrong answer" even if the program works on the student's computer. Or the character-by-character equivalence in text output files causes errors if real numbers are rounded off or truncated. These unexplained errors trigger confusion and frustration among students. Moreover, none of these studies gave information on the alignment of instructional activities prescribed in the DSA coursework or of the underlying pedagogy used for assessing coding practices.

Belhaoues et al. (2016) caution on the need of semantic structure and pedagogical approach in using algorithmic exercises for DSA coursework. Explanation, representation and formalization of the algorithmic exercise must be expressed clearly for learners to act on a given problem. Moreover, an underlying pedagogical objective is crucial when AA tools (e.g., Algo+) are to be used for delivering programming exercises to students. That is, the knowledge base (or algorithmic exercises base) must have proper ontological grouping that clearly puts forth the theoretical notions of the domain coupled with a rational approach to support students in responding to exercises that are then graded by AA tools. Description of each exercise put forth via the AA tool must have a justified pedagogical foundation that formalizes a set of skills and notions pertinent to that algorithmic field of study. The authors note that while plenty of algorithmic problems may be available in public databases, they often lack pedagogical, semantic and epistemological organization. Each exercise activity should have a student-centred focus for building up of students' knowledge skills (i.e., know-what, know-how and know-why skills).

Soll et al. (2021) describe the use of CodeRunner, a readily available AA tool, that can be embedded in the Moodle learning management system (LMS). They created relevant programming exercises (based on DSA coursework) and allowed students to submit their code to CodeRunner from their LMS login accounts. While students could apply their knowledge gained in the lectures to the programming tasks, many problems were encountered with CodeRunner. These included lack of transparency of the user interface, absence of debugging tools, long execution time whenever infinite loops were encountered and other malfunctions during program execution (e.g., errors when dealing with corner cases, missing test cases for unanticipated errors, etc.).

## Purpose of this study

Published literature has highlighted the significance of AA tools specifically in the context of DSA coursework. Prior studies lay emphasis on having a pedagogical approach when using e-assessment tools (e.g., Mooshak (Gárcia-Mateos & Fernández-Alemán, 2009), CodeRunner (Soll et al., 2021)) for integrating programming exercises into the coursework. Further questions are raised regarding the instructive value of available off-the-shelf AA tools (Gárcia-Mateos & Fernández-Alemán, 2009), as these are seen to be lacking in their ability to give proper formative feedback to students, besides having other technical difficulties (e.g., strict file formats, infinite loop traversals, different compiler versions, etc.). Taking these shortcomings into consideration, we propose a bespoke automarker specifically designed as an instructional tool for formative learning activities. Activity Theory has provided the underlying pedagogical lens (Basharina, 2007) for analysing learner interactions with the prescribed coursework activities. The fundamental concept of AT is the 'activity', which in this case study, relates to seven instructional activities pertaining to different topics/sub-topics of the DSA coursework.

The study demonstrates the appropriation of the automarker as an instructional strategy for experiential learning and building learners know-what, know-how and know why competencies. We describe how our automarker e-assessed students' programming assignments and overcame the technical difficulties identified in previous research studies. Further, we explore if students could meaningfully engage with the enumerated coding tasks to complete their assignments (know-what) while working in the given automarker-mediated settings. Also, how did the automarker-mediated feedback assist students in enhancing their logical and reasoning capabilities and help them in successfully accomplishing the given tasks (know-how). And finally, did these ongoing interactions enable self-reflection, wherein students corroborated the automarker's feedback and analysed different coding perspectives for further advancing their algorithmic reasoning skills (know-why).

## Activity theory

Activity theory has enabled reflection of pedagogical practices by zooming into different learning activities which are goal-oriented and tool-mediated within some higher educational context (Mathrani et al., 2020; Murphy & Rodriguez-Manzanares, 2008). Daniels (2004) views that AT has opened up our pedagogic imagination as it can facilitate empirical research on pedagogical use of digital technologies with specific analysis of how different activities enable transmission of the prescribed knowledge and skills. For instance, Park

and Jo (2017) used AT to evaluate students' usage patterns from log data extracted from an institutional learning management system. While Adam et al. (2019) examined student perspectives of how online tools are implemented in virtual learning spaces in a developing country context. Activity theorists can therefore investigate complex human–computer interactions by breaking them down into smaller categorical elements to understand purposeful acts of individuals as they work within some constraints with appropriated tools (Basharina, 2007).

AT extends the three elements of the Vygotskian triangle (1934–1987) into six elements that together realize the final outcomes. The three elements of the Vygotskian triangle comprise *subject* (instructor of the DSA course), *tool* (automarker) and *object(ives)* (for building algorithmic reasoning and programming skills among learners). The expansion of the triangle with three more elements, namely, the (learning) *community* (comprising instructor and students), *rules and regulations* (prescribed in the DSA coursework) and *division of labour* (spanning teacher and learner roles) provides the case background. Further, the core of AT is the proposed learning *activity,* which emphasizes how ongoing interactions are taking place between the six elements. The (learning) community members have clearly defined responsibilities specific to their role as a teacher or a learner and each must comply with set rules and regulations for meeting desired goals. Pedagogical use of digital technologies (e.g., automarker, test cases, LMS, etc.) further assist in achieving learning goals, which are then transferred into final *outcomes*, such as, facilitating programming practice among learners and establishing effective workaround strategies for the instructor. Figure 1 (A) and (B) demonstrates the underlying AT elements and how these are mapped to this study's context.

Computing courses are interleaved within different types of technology-supported environments where learners immerse themselves with ongoing learning experiences. The teacher (*subject*) prepares practice learning scenarios to improve students understanding of technical knowledge (e.g., data structures, algorithms, network applications), and together they form (learning) *community* spaces. Within these communities, students implement a wide range of learning activities that are governed by the *rules and regulations* laid as per DSA coursework as they acquire new knowledge (i.e., know-what, know-why and know-how). The *division of labour* refers to teacher's responsibilities in the preparation and presentation of appropriate formative learning exercises for facilitating student engagement. Since different programming skills are applied to different topics/sub-topics, the teacher must plan instructional exercises that are pertinent to each topic. Teachers, set

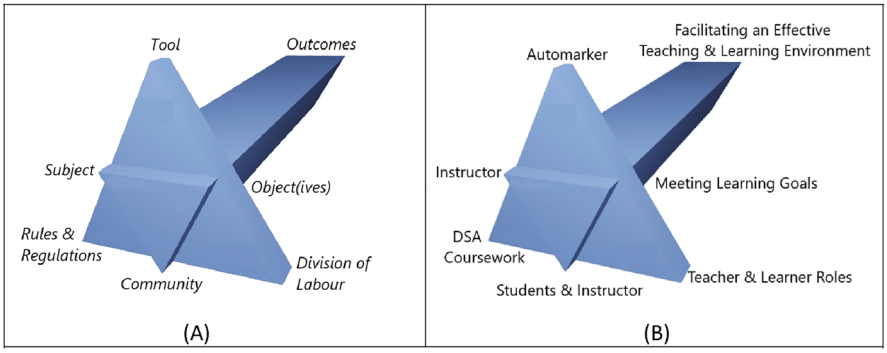


**Fig. 1** Activity theory contextualized to this study's context

up practice scenarios which require students to submit their code to be assessed by the *tool* (automarker). For example, a scenario could be related to sparse matrices or to queue implementations. The tool instantly checks each student submission and displays the marks to the student. In case of any mistakes, it presents them with clues to rectify these mistakes and further provides more opportunities to re-submit and improve their marks.

In doing so, *four objectives* or goals are met. First, the teacher can dynamically display all coursework exercises which can be leveraged by the enrolled students. Second, students can independently engage with the tool and practice their coding skills where they can self-verify the correctness of their code and make emendations before making a final submission. Third, the time taken by the teacher to manually evaluate each student code submission and provide them with personalized feedback is eliminated. Fourth, any chances of errors that can accidentally crop up with manual marking process are much reduced. The accomplishment of these goals further informs the final *outcomes* in setting out clear expectations regarding student feedback and assessment style for facilitating an effective and interactive teaching and learning environment.

## Research design

Research about the pedagogical use of digital technology spans the development of the technological system in a real-world teaching context; therefore, any empirical data that is gathered is symbolic and must be interpreted in its appropriate context (Twining et al., 2017). Twinning et al. advise researchers to properly outline their research design by providing rich descriptions of their research setting, the data collection instrument and sample size. A copy of the data collection instrument may be provided either as an appendix or as a linked document for adding more context (Tong et al., 2007). Such background information allows the reader to understand the research setting from an ontological position (or the nature of reality). In the field of instructional technology, the researcher must consider the extent to which their product's design can be considered a specific intervention or is generalizable to a target population (Richey et al., 2004). Our instructional product (automarker) comprises learning activities that focus on self-evaluation by students as they build their programming skills; hence, is generalizable to student populations pursuing programming coursework. These learned skills can enhance students' ability to tackle a wide range of programming scenarios and help develop a problem-solving mindset, all of which are applicable across various courses of computer science and beyond. Richey et al. (2004) recommend that generalizable products should clearly specify the design, implementation and evaluation methods that have been used. We have laid out the architectural design of our proposed automarker tool and how it has been deployed in a real-world teaching and learning environment in the following two sub-sections. Next, in the subsequent section, we have described how the tool framed various coursework activities and how the learner community comprising of both students and teachers interacted with it. Subsequently, we conducted a student survey to understand how the various learning interactions facilitated by the tool helped build learners' programming competencies. A copy of our survey instrument is given in Appendix A. Student participation to the survey was voluntary and anonymous. Our survey was conducted over two consecutive DSA course offerings, for which we received a total of 46 responses from a combined class size of 100 students. Of these, seven

responses have been eliminated, as these comprised blank responses, with neither any rankings nor any comments. Therefore, 39 valid survey responses have informed our study.

Further, Twining et al. (2017) add that the underpinning theory too needs to be explicitly articulated so that the epistemological position (or the nature and scope of knowledge) is clear and enables the researcher to provide justification of their results and of the conclusions drawn. This will ultimately demonstrate "how we come to know the world" and showcase their interpretive judgement (p. A2). Activity Theory (elucidated in the previous section) has framed this research study and helped to interpret the role of technology (i.e., the automarker) for building knowledge competencies among students with the use of practice-based programming exercises. Section "Viewing from the activity theory lens" describes how AT worked as a pedagogic lens for viewing the alignment of the automarker with formative learning exercises that were put forth for a given coursework. This alignment is explained in the context of the learning community comprising students enrolled in a DSA course and the teacher of the course.

The following sub-section outlines the architectural design and deployment of the automarker for DSA coursework. The subsequent sub-section expands on how the automarker's software design helped to overcome some of the limitations of available off-the-shelf AA tools that are mentioned in prior literature (and which have been elucidated in Sect. "Related works").

### Automarker: architectural design and deployment

The automarker has been purpose-built for mapping various programming exercises (in C++ language) prescribed for a 2nd year DSA coursework of an undergraduate degree programme. It comprises a client–server architecture, where the code is written in PHP and all students' data is stored in MySQL. The deployment uses Apache under Linux. The server exists as a stand-alone system that is not integrated in the institutional LMS; hence, access to this server could be availed either from within the institutional network or via the institution's virtual private network (VPN). All students are eligible to apply for VPN access from their institution. Provision for VPN access is especially relevant for CS courses as these are known to make extensive use of dedicated server configurations for practice teaching and learning purposes.

Students upload their C++ code files (based on prescribed programming exercises) to the automarker using a code template (or code skeleton). The server immediately compiles student's code using GCC (Stallman & DeveloperCommunity_Gcc, 2009) after which it executes the code over 10 test cases, while referencing a model solution supplied by the teacher. Depending upon the number of test results that matched with the model answers, the student files are marked anywhere between 0 and 10. Students are given endless opportunities to resubmit their codes multiple times until the assignment deadline is reached. Submitted code is automatically evaluated using a set of inference rules (via test cases) and feedback is provided by means of clues which in turn encourages students to debug their code and improve their marks. There is always the danger that fixing code to get one of the tests to present the correct answer may break a test that had previously passed. This is all part of the learning and self-reflection, as students strive to validate their code for all tests. Therefore, with this form of assessment design with the automarker tool, instructors can instil algorithmic concepts in students, as they encourage students to hone their programming skills through sustained practice. Figure 2 illustrates the flowchart of the students' code submission process.

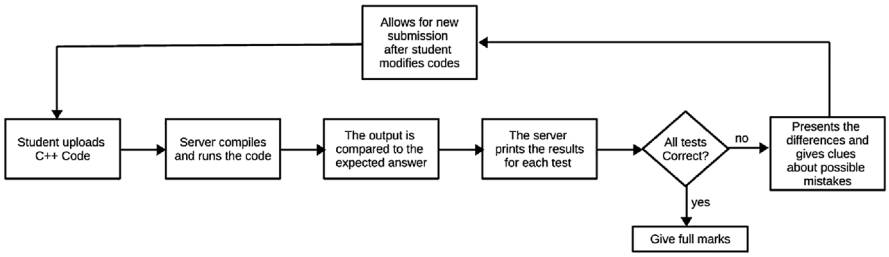


**Fig. 2** Flowchart of the submission process

## Automarker: software design

AA tools are known to be constrained by precise formatting of files, such as character-to-character equivalence across text files, since assignments are directly handled by the machine; therefore, having a well-defined code template is crucial (Rubio-Sánchez et al., 2012). While some formatting instructions were laid out to the students regarding their code submissions to the automarker, we have added some more flexibility. In our case, a simple parser allows for some text variation, such as spaces and capital letters, in reading results from the given code for each test input. Additionally, specific values expected from the test results have been considered within a certain interval, thus enabling the teacher to set up an interval that they consider close enough to the quasi-optimal answer. This has provided a huge advantage over off-the-shelf AA tools, since it helps overcome the marking errors that could occur with different compiler versions that students may have used, or when dealing with certain optimisation problems where the answer may be only an approximation of the optimal answer. One such example is the queue assignment, where the results can depend on the initial state and on the order of the updates of multiple queues. Correct answers may vary a little for slightly different solutions found by students. Since the automarker can be set with a tolerance value, each test will consider this tolerance to inform students about the result of the test cases. Those assignments that have only one correct answer per test have their tolerance set to zero.

Another advantage offered by the automarker is that students need not be connected to the server all the time, rather they can develop and test their codes in their own environments. Students only connect to the server when they have pre-tested their code in their machines. This is a major problem exposed in the study by Soll et al. (2021), where they used CodeRunner that was implemented in Moodle. The study found that students often blamed the system for problems in their code. A typical complaint was that their AA system was not giving students enough time to run code, when in fact students had an infinite loop. Our automarker gave specific amounts of time to run the code, and if it ran out of time the feedback specifically stated the possibility of an infinite loop in the code. It is not possible to know if the program really is or is not in an infinite loop, due to the undecidability of the halting problem (Davis, 1958). However, one can still infer that if the program took more than 20 or 30 s when it should have taken a fraction of a second, it is either because the code is extremely inefficient or goes into an infinite loop. In either case, students must modify their code to fix the problem. Moreover, as students could also run the code in their own machines prior to submitting the code to the automarker, they could easily find these problems.

Another major limitation in AA tools is that it is not possible to analyse the code directly and give a specific clue as to why the code is failing. This is part of life of a programmer, as they need to learn how to code, and more importantly, learn to debug their code. As part of the course, we also teach debugging techniques, and try to convey the message to students that they are ultimately responsible for their bugs.

## Viewing from the activity theory lens

Engeström (1999) suggest that analyses of activity-theoretical constructs can lead to framing of multiple perspectives, such as 'mental model', 'repertoire', 'social representation' or 'attitude', which are manifested in how learning is undergoing developmental transformations over time. AT provided us with a technocentric lens (Murphy & Rodriguez-Manzanares, 2008) to examine the prescribed DSA coursework tasks and understand the (learning) community perspectives in regard to how the instructional activities led to the transformation of the goals/objectives into the final outcomes in the automarker-mediated environment. These two perspectives are elaborated next.

### Coursework perspective

The learning principles in CS are strongly practice-driven with "40% of its core hours [allocated to] algorithms and complexity, programming" and building "abstract computational capabilities" (Computing_Curricula_, 2020_Task_Force, 2021 p. 27). Therefore, the DSA coursework was organised into seven distinct topics related to the different data structures, namely linked-list, stack, queue, list, tree, graph and heap. Algorithms pertaining to appropriate application in the conduct of additions, conversions, searches, sorting, and evaluations within some given problem context (e.g., find the shortest path, count the number of operations, etc.) were put forth. AT describes activities as an empirical social construction process that involves goal-directed actions and motive-oriented actions which are performance based. The topics informed the design of the (learning) activities that gradually extended in complexity of programmatic concepts and their applications to challenge students in building their knowledge capabilities.

Table 1 describes the seven learning activities prescribed in the context of the DSA coursework and lays out the functionality of the automarker for marking of the student code submissions. The automarker evaluates the performance of the submitted codes according to a list of programming tasks required for accomplishment of learning activities and enables learner interactions via a fusion of know-what, know-how and know-why competencies.

The automarker employs a variety of tactics and tools to mark the assignments accurately. Firstly, it imposes different levels of stress tests to gauge the robustness of the submitted code solutions; thereby, allowing it to award scores that commensurate with accuracy of code solutions. It consults a model answer key to check if the output result from the student code solution matches within some tolerance interval. Expert knowledge of the subject content is infused into the marking strategy. For those items/tests where there is exactly a single correct answer (e.g., minimum path cost), a tolerance of zero is set. On the other hand, for tests where slight variations in code implementations are expected to generate slightly different performance outcomes (e.g., count of the number of operations), a tolerance interval is set accordingly. The use of a timer as a guard against infinite loops

**Table 1** Activities for data structures and algorithms coursework

| Activity | Coursework description | Tool functionality related to coursework |
|---|---|---|
| 1 | ***Linked-list*** *implemented in C:*<br>Add two sparse matrices | Verify the final matrix<br>Check for infinite loops<br>Check for correct representation of sparse matrices<br>Check for the printout of the linked-list (order of elements) |
| 2 | ***Stack*** *implemented as a C++class:* Evaluate RPNs using stacks | Verify the evaluation of the RPN<br>Check for "too many numbers" or for "too many operators" errors<br>Check for infinite loops |
| 3 | ***Queue*** *implemented as a C++class:* Find the maximum number of packets in a router using queues | Verify the final peak congestion of the traffic<br>Check for time, if too long issue a possible infinite loop message, or too slow due to incorrect implementation of the queue methods<br>Check result for large simulations that include 128 ports |
| 4 | ***List*** *implemented as a template class:* Add two arbitrarily large integers represented as a list of digits | Verify the final result<br>Check if it works numbers that have different lengths<br>Check the results of the sum for every test |
| 5 | ***Tree*** *implemented as a class, supported by a Stack of Tree*:* Convert an RPN expression to an arithmetic tree using stacks | Verify the in-order and post-order printouts<br>Check for infinite loops<br>Use large RPN to see if the code still works |
| 6 | ***Graph*** *implemented as a class:*<br>Find the shortest path from a source node to every other node in a graph (Dijkstra algorithm) | Verify the minimum distances to each node<br>Check for infinite loops<br>Use graphs with islands to stress the code<br>Check final minimum distances and verify that they are not infinite |
| 7 | ***Heap*** *implemented as a class. The Heap is implemented with a vector (STL):*<br>Using Heap sorting, count the number of operations to insert and delete elements of Heap | Each test is carried out with a randomised sequence, an inverted sequence and an already sorted sequence<br>Verify that all three files were correctly sorted<br>Verify the counter for each Heap operation<br>Use file sizes from 50 to 5000 |

enables the automarker to check codes in extremely challenging scenarios. Moreover, the same timer mechanism also enables the automarker to catch highly inefficient and incorrect code implementations, as such codes would run significantly slower than the correct ones.

### Community perspective

Students enrolled in the DSA course, and the teacher of this course form the learning community. In presenting the seven coursework activities, proper care has been taken to provide a consistent and simple user interface so as not to distract the students with unnecessary detail. The automarker ensured personalized interactions with the learner are maintained to retain learner's interest, such as by providing them with some clues on likely mistakes in their code. In this manner, the automarker provided guidance (in lieu of the teacher) leading to self-awareness as students could self-assess their code and work towards fulfilling their assignment expectations.

### Student perspective

Students engaged with the automarker to self-assess their C++ code files (that formed part of their assignment) before making a final submission of the assignment for grading. Figure 3A presents examples of the feedback provided when all test results are correct, Fig. 3B shows an example where the first test has failed errors while Fig. 3C shows a mix of pass and fail tests.

The automarker's parser allows for minor differences between the standard results and student's responses, as long as the output gives the correct numerical or logical answers. One of the frustrations with other AA tools is that a single space or minor formatting difference will trigger a test failed response even though the correct result is outputted. The automarker on the other hand gives a warning in case of format differences; however, if the test result is numerically correct the marks are given to the student.

The automarker is a generic tool for any kind of assignment, hence the answers cannot be hardcoded for any activity. Instead, the parser shows the difference between the expected result and the student's result for each test case. The student can look at the display to understand how to arrive at that result. Some clues are provided, which are not very specific, rather they align with the learning concept, that is, the student is given the clue in the context of the assignment to enable some understanding on the difference. Moreover, normally half of the inputs are known to the students, but the other half are kept hidden. This is to avoid student attempts to hard-code their results to fit into a certain pattern. Moreover, the hidden tests are usually too big to allow for hard coding. For example, in the sorting assignment, a certain data structure called Heap needs to be created that is used in the sorting algorithm. Some of the tests involve thousands of numbers and so it is not feasible to create the result by hand. The logic must be therefore understood with a small example, which can then be generalised to larger examples.

Another aspect of the automarker is that specific tests can be made more difficult to pass; accordingly, the students need to generalise their code for all cases. In the example shown in Fig. 3C, the student's code passes when the RPN expression (reverse Polish notation) is correct, but it is unable to display when the RPN is incorrect. This could be because there are either too many numbers, or too many operators. This gives a chance to the student to consider all such results when modifying their code. Also, a student may get partial marks, in which case they are rewarded for the efforts made so far.

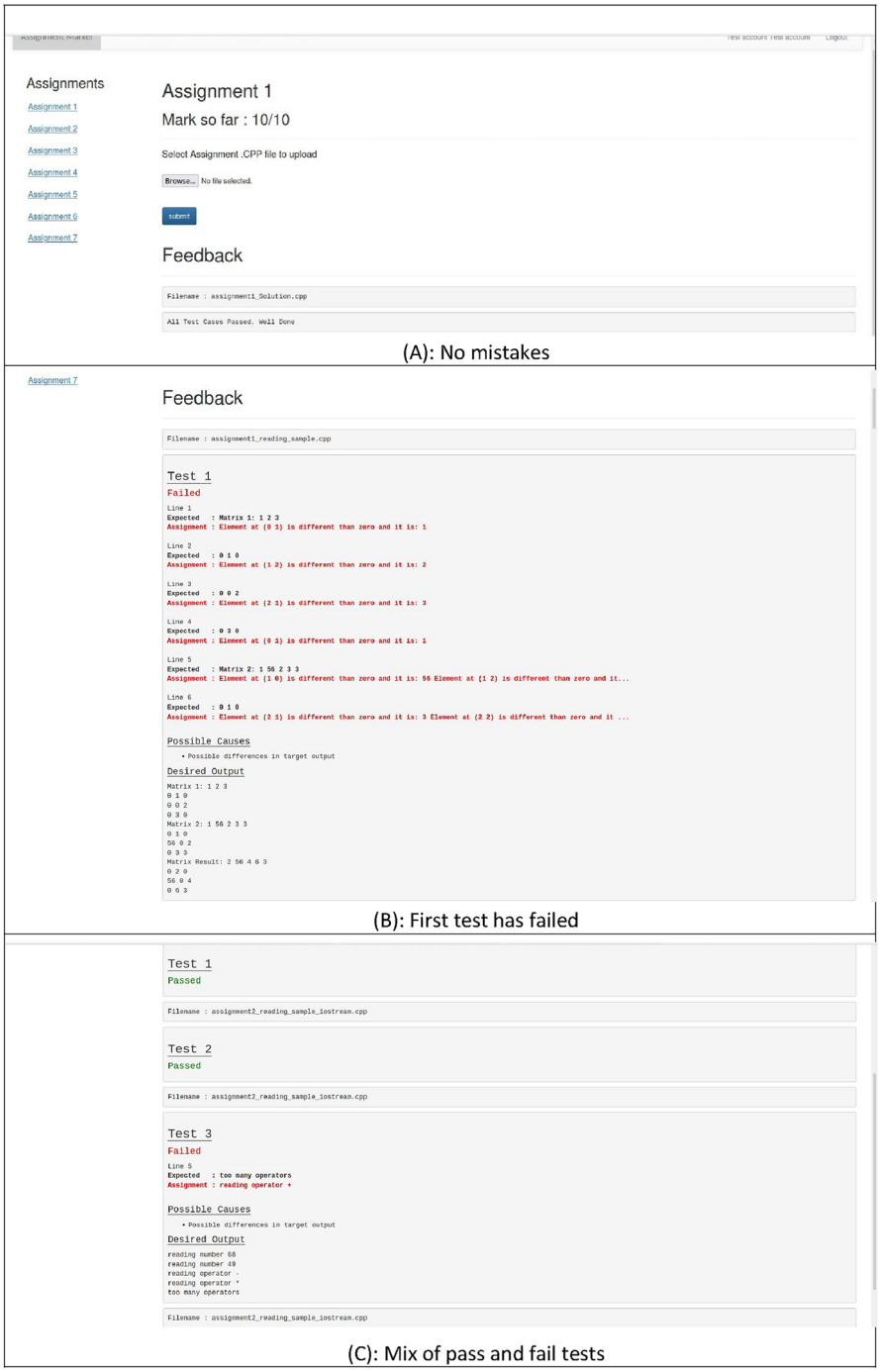


**Fig. 3** Automarker displays for students

### Teacher's perspective

The teacher's first responsibility is creating assignments that align with the coursework. With the automarker, the teacher can create as many assignments as they want, since deploying a new assignment requires little set up time. Once a new assignment is created, the teacher runs the model solution with given test data input and saves the output result to a text file. Simple text files hold the input (or inputs). Students are taught how to get arguments from a command line over the coursework where they use text files as the only input for their assignments. The format of these input files is explained to the students when the assignment is first presented. Once the instructor has both input files and output files for all test cases, then these files are uploaded into a specific directory in the server using the teacher's login account (that has additional privileges).

The automarker offers other administrative functionalities. For example, before a course starts, the instructor uploads the names of all enrolled students (via a.csv file) using the administration screen. Students accounts are then automatically created in the server (which includes this information in the SQL database), and further directories are created which will contain the test files. The students are emailed individual login accounts (comprising name and password). Moreover, single addition or deletion changes too are supported, for example, to accommodate a late course enrolment or a case of an enrolment cancellation.

The administration screens also allow for parameter setup in assignments (Fig. 4A), such as number of assignments or tolerance for the numerical results. A single screen view (Fig. 4B) presents all the current results for each student (anonymised). This is a good feedback stream to the instructor since it informs them how students are coping with a

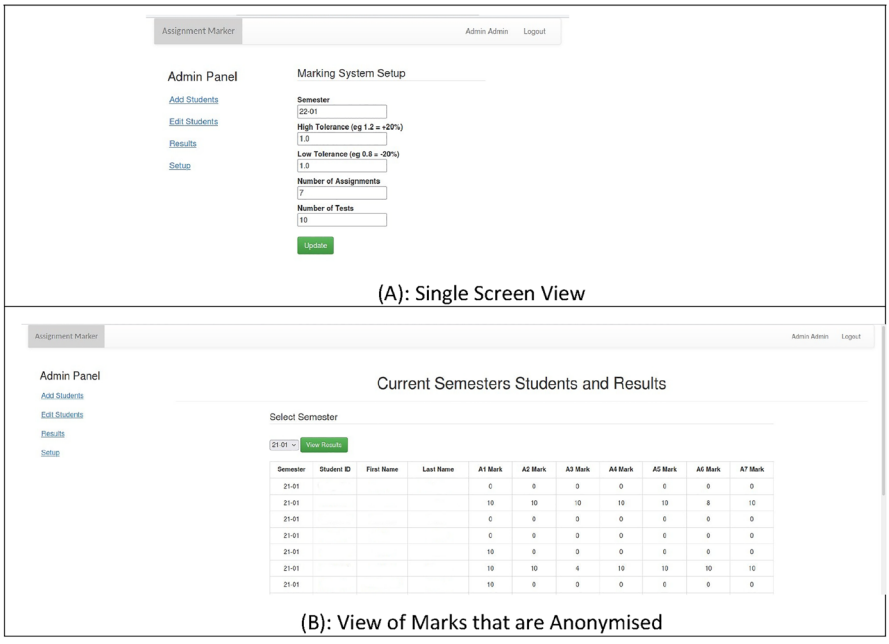


**Fig. 4** Automarker displays for teachers

particular assignment; this is especially helpful for assignments that have more complexity. The teacher can then offer extra explanations or provide help in such assignments as soon as the problem is identified. Often a lack of pre-requisite courses can lead to learning difficulties; therefore, such feedback can assist the instructor in identifying this to be an issue for a certain assignment, based on which they can provide additional support.

The instructor can re-visit student codes and check the results if needed, without any specific software or tool. Further, the instructor can re-run the student codes as many times as they need to, and the results of their re-runs will not interfere with the student's submission. From the instructor's point of view, the automarker provides them with much flexibility, as it only requires them to run test inputs using the correct code to generate text files that can be used directly by the server. It is also very easy to create new assignments, generate new test cases or modify the format of the output with minimal effort, and make assignments instantly available to students.

## Discussion

This section consolidates how the learning interactions aided in building know-what, know-how and know-why competencies with the commissioned automarker. We draw upon the evidence gathered from a student survey that was conducted in the latter part of the teaching semester (i.e., after five assignments). This way, survey responses were based on students' current emotional state and were not retrospective in nature.

The first question (on the first page of the survey instrument) asked students to rate their C language programming skills before enrolling in the DSA course. The DSA coursework used C++ programming language which had a pre-requisite programming course on C language. The question posed was: "How would you have rated your C language programming skills before enrolling in the DSA course (i.e., writing, compiling and running simple C programs using arguments from command line)?" Self-assessment was considered as a useful starting point since it allowed students to first reflect on their programmatic skills prior to enrolling in the DSA course. Students ranking were evenly split with 19 responses marked as good/excellent and 20 responses as average/poor (Fig. 5).

Next, we asked them how satisfied they were with the instructions provided, the format of code templates and the sample files used for file submissions. We also queried on the ease in creating and submitting code files to the automarker. These helped to gauge the know-what aspects, that is, could students understand the step-by-step instructions regarding the file formatting and whether they could create such files easily. Overall, students

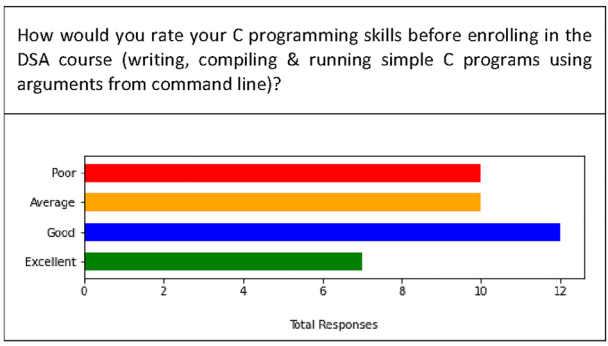


**Fig. 5** Programming self-evaluation by students

expressed their understanding as satisfactory and said that creating files in the specified format was an easy task with only one response stating this as not easy.

The following two questions sought to understand whether the clues and test cases enabled students in understanding their coding errors and their level of satisfaction with the feedback provided. That is, could students transform the feedback into a higher-order skill level and independently resolve their coding errors (i.e., know-how). Students overwhelmingly stated that the clues/comments were extremely helpful in completing their assignments (30 responses) though few others considered it moderately easy or were neutral. Regarding their level of satisfaction with the feedback from the automarker, responses ranked this as very satisfied, moderately satisfied or neutral, although one respondent gave it a dissatisfied rank.

Finally, we asked students whether the provided feedback (from the test cases) helped in code debugging and further how actively they worked towards passing all the test cases. These self-reflection questions gauged whether students had tried to fully grasp the feedback which was then actioned thoroughly into their final code submission (i.e., know-why). Students gave positive responses to both these questions with 37 responses indicating that they had actively engaged in resolving coding errors based on the automarker feedback, with one student saying that he did not make full use of this feedback, while one other had never used the automarker. Figure 6 provides a snapshot of all these survey responses, which have been categorized into know-what, know-how and know-why skill measures.

Next, students had to rank all the five assignments in the context of learning of programming constructs, their level of interest and overall experience with automated assessments. Responses reveal that majority of students were much satisfied. Overall, students considered the assignments had assisted them with learning programming,

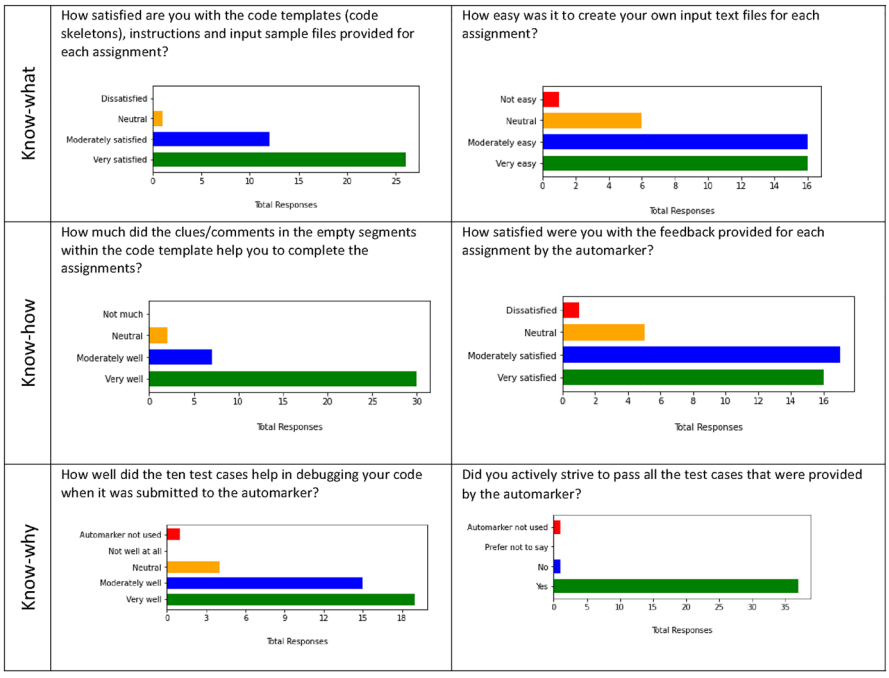


**Fig. 6** Quantitative measure for know-what, know-how and know-why skills

had made them interested in the different programming tasks and that this form of assessment was an excellent way to learn programming. Based on these responses, we find that the automarker served as a formative instructional strategy for experiential learning and honing students' programming skills.

Next, students were asked to provide (optional) comments while ranking their responses. While students indicated much positive aspects of this form of e-assessment, their comments provided more context to their ranked responses. One student observed "*some assignments are too easy, due to the amount of skeleton code*", while another said that another student said that "*the automarker takes away the guessing game for assignments so it's easier to know how well you've done in it. I am a big fan of it"*. Another student found the template used "*too much C shorthand* [which had] *no hanging brace format*" which can get "*confusing*" as these "*students have only completed 101* [i.e., the first-year pre-requisite course in programming]" programming course.

Regarding know-how skills, students said that the use of test cases and clues enabled them to correct their coding mistakes and re-submit their assignment. One student commented "*It's good to know where code is expected but don't give the game away*" implying that sometimes the feedback was rather explicit, which made it an "*easy*" task. Yet another student voiced *"passing of all test cases would give me confidence that my program works as intended for a variety of test cases"*. Other responses were *"overall I think this assessment model is an improvement"* and *"it's made debugging easier"*.

In response to the know-why questions that enabled self-reflection, one student said "*actually it is hard to know where the bug is. I have to think about it myself*". Another student added that their code still had bugs and they "*did not get all pass result*" in some of the programming tasks. Another response said that they needed more such assignments *"to cover all the algorithms taught in the course"* while another added that they had learned *"about how to debug and find errors in the code"*.

One student voiced irritation about access to the VPN. Students must specifically request for VPN facility from the host institution and set it up on their home machines. Those students who had not availed the VPN facility found it *"annoying"* that their C++ code could not be checked from within their LMS (Moodle). Workplaces often require workers to use VPN for accessing organizational systems; therefore, having awareness of VPN protocols is also considered a learning activity. The student commented that they did not use the automarker as they faced difficulties in accessing it from outside the institution's network which was "*annoying*". No mention of reasons on why the VPN facility was not availed was made. However, many students found this type of formative assessment useful, especially since it gave them immediate feedback. This one comment "*the automaker really helps and every programming course should have this*" sums up the student satisfaction levels, which is also evident from quantitative rankings given by the students (Fig. 6). We find that overall response towards the use of the automarker was favourable, with only one student expressing their dissatisfaction.

Figure 7 provides more self-evaluation for the five programming assignments. Student responses indicate that formative assignments when facilitated with AA tools helped students learn relevant programming constructs and largely enhanced their interest in programming. Overall, students found this form of e-assessment with automarkers an excellent learning experience. However, few students were neutral and did not express much appreciation for the automarking strategy used in the DSA course.

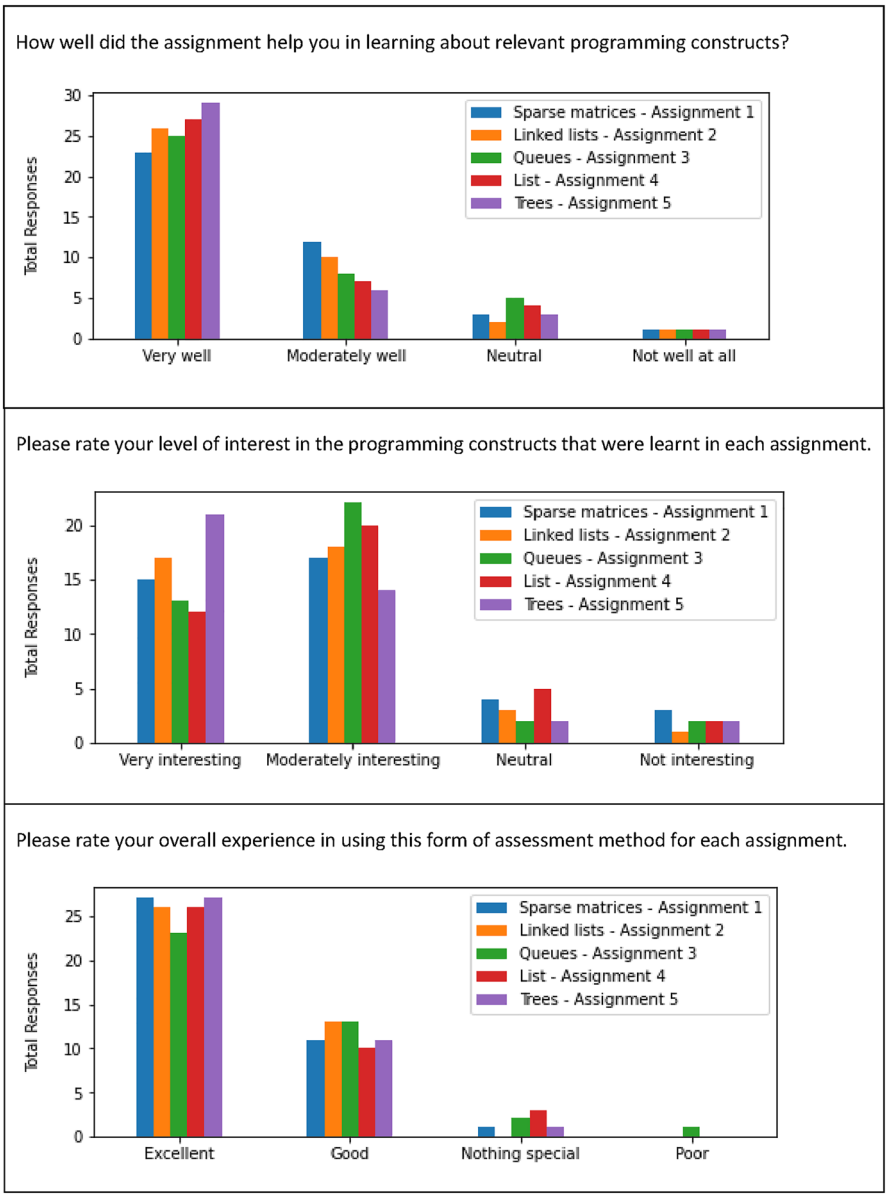


**Fig. 7** Ranking of programming activities (sparse matrices, linked lists, queues, lists and trees)

## Conclusion

Educational pathways in practice-oriented disciplines, such as computer science, call for coursework that promote experiential learning and support students in building programming abilities that demonstrate a higher-order skill level. Computer science courses involve continuous practice tasks with frequent formative exercises so that students can attain appropriate

knowledge competencies that are much in demand by tech businesses of today (Zvacek, 2015). However, manual assessment of ongoing formative exercises (especially programmatic tasks) can become very time-consuming and mundane for teachers, more so, with the high student-to-teacher ratios in the computer science discipline. Therefore, automated assessment tools can assist in frequently conducting formative assessments while at the same time not overburdening teachers with marking and grading. Moreover, the strategy of allowing students to resubmit codes multiple times and provide them with automated feedback encourages them to practice and build their coding competencies to develop a higher internal mental functioning.

A detailed analysis of different learning activities prescribed for a foundational course, Data Structures and Algorithms, has revealed how e-assessment strategies can bring about self-awareness and self-reflection among students. Activity Theory helped frame learning interactions with our proposed tool as we investigated e-assessment as an instructional strategy for formative learning. The tool helped students deliberate on what programming tasks had to be carried out (know-what), assisted them in prompt resolution of programming errors that may have been made (know-how) and engaged them in higher-order thinking as they reflected over multiple programming tasks of varying degrees of difficulty (know-why). As students developed self-awareness after self-validating their code, it helped them achieve sustained practice in writing programming code effectively (Barra et al., 2020).

We have provided substantive focus on the nature of the tool design, what it represents in terms of the prescribed coursework (pertaining to Data Structures and Algorithms) and how this can be used for practice-based assessments. It has further enabled the teacher to produce frequent assessments without having to take on additional load of marking/grading the assessments. All this contributes to clarifying the ontological position or the nature of reality in educational settings. Next, the epistemological position is articulated with the framing of pedagogical elements with the Activity Theory constructs. The automarker served as an instructional tool that was integrated into course-specific formative learning exercises. As students interacted with the tool, they were provided with relevant clues/comments to help develop their practice of writing effective code. Ongoing negotiation with the tool further helped in the construction of new knowledge across the community of learners. This emergent knowledge is very much practice-driven based on student participation with ongoing learning activities until higher order levels of skills and competencies are developed. The three skill levels articulated in the latest computing curricula (2021) have been fortified by a student survey. Survey data reveals student perceptions towards their know-what skills (i.e., their understanding on the subject matter and in following the given task instructions), know-how skills (i.e., their ability to interpret the feedback provided and applying it to improve the quality of their code) and know-why skills (i.e., their inclination and disposition to "connect the 'better' or 'correct' application of knowledge and skills to the context where and why it is applied" (p. 48)). Our findings show that the automarker supported student learning to actively pursue their programming tasks and guided them in reaching higher-order cognitive skill levels. Such an automated instructional strategy can have implications for learning other programming languages, or in other practice-based course curricula settings.

## Limitations of the study and future work

This study has provided empirical evidence on the use of e-assessment as an instructional strategy during formative learning to enhance the overall learning experience. We acknowledge that the survey data is limited to thirty-nine student participants and therefore

offers a narrow view on how our automated assessment tool aided students in experiential learning to develop their programming competencies. Moreover, this study was restricted to empirical data gathered from student surveys and no student interviews were held. We were bound by the ethics approval guidelines specified by the institution, which laid the clause of student anonymity since one member of the author team had designed the tool. Interviews would lead to disclosure of students' identity to their instructor and could have caused some discomfort to the students in freely expressing their opinion. Therefore, we used both ranked and open-ended survey questions which did not compromise on student anonymity. Moreover, with this tool being used in a real-world course delivery within a university environment, we offered the tool to the whole cohort enrolled in the said course, hence it was not possible to make comparisons between control and treatment groups. Overall, based on the data collected from students' self-evaluation, it is clear that the proposed system provided an excellent learning platform for the whole student cohort.

This study has offered new insights on how an automated assessment can contribute to both curricula and pedagogical practice. The curriculum in this study corresponds to the use of formative exercises in a C++ programming course (Data Structures and Algorithms) and pedagogic representation shows meaningful integration of an e-assessment tool to support learners and develop their knowledge competencies. Based on our experience with the automarker, as part of future work, we envision that we could also employ a more proactive strategy in motivating students to work on improving their programming competency, rather than waiting for them to make mistakes. Further, to help characterise the student engagement and learning curve in more detail, a system facility that could keep track of their frequency of use, improvements, mistakes committed, number of attempts per task, time of first access relative to submission date, areas needing improvement, etc. would be useful.

Lastly, it leads the authors to believe that there is merit in applying the same system to other remote asynchronous courses, but that would require further investigation as part of future work. We are also working on integrating the automarker tool within the current e-learning system at the institution, as that will provide a single-entry point to students and remove the requirement of using a virtual private network.

## Appendix A

Survey Instrument

| | |
|---|---|
| Page 1 | How would you have rated your C language programming skills before enrolling in DSA (i.e., writing, compiling and running simple C programs using arguments from command line)?<br>☐ Excellent ☐ Good ☐ Average ☐ Poor |
| Page 2 | How satisfied are you with the code templates (code skeletons), instructions and input sample files provided for each assignment?<br>☐ Very satisfied ☐ Moderately satisfied ☐ Neutral ☐ Dissatisfied<br>Extra comments (optional) |
| Page 3 | How easy was it to create your own input text files for each assignment?<br>☐ Very easy ☐ Moderately easy ☐Neutral ☐ Not easy<br>Extra comments (optional) |
| Page 4 | How did the clues/comments in the empty segments within the code template help you to complete the assignments?<br>☐ Very well ☐ Moderately well ☐ Neutral ☐ Not much<br>Extra comments (optional) |

| | |
|---|---|
| Page 5 | How satisfied were you with the feedback provided for each assignment by the automarker?<br>☐ Very satisfied ☐ Moderately satisfied ☐ Neutral ☐ Dissatisfied<br>Extra comments (optional) |
| Page 6 | How well did the ten test cases help in debugging your code when it was submitted to the automarker?<br>☐ Very well ☐ Moderately well ☐ Neutral ☐ Not well at all<br>☐ Automarker not used<br>Extra comments (optional) |
| Page 7 | Did you actively strive to pass all the test cases that were provided by the automarker server?<br>☐ Yes ☐ No ☐ Prefer not to say ☐ Automarker not used<br>Extra comments (optional |
| Page 8 | How well did the assignment help you in learning about relevant programming constructs?<br>☐ Very well ☐ Moderately well ☐ Neutral ☐ Not well at all |

This was in a grouped question format that applied to the 5 assignments (Assignment 1-Sparse Matrices, Assignment 2-Linked Lists, Assignment 3-Queues, Assignment 4 -Lists and Assignment 5-Trees

**Acknowledgements** We thank the students who voluntarily participated in the survey of the effectiveness of the automated assessment tool used in this research.

**Author contributions** ALCB: Conceptualisation, Software, Formal analysis, Investigation, Visualisation, Writing original draft, review and editing. AM: Conceptualisation, Formal analysis, Investigation, Methodology, Visualisation, Writing original draft, review and editing. BH: Investigation, Writing—review. NR: Revision, Software, Writing—review.

**Funding** Open Access funding enabled and organized by CAUL and its Member Institutions. No specific funding was obtained for this research, not applicable.

**Data availability** Not applicable.

## Declarations

**Conflict of interest** The authors declare that there are no competing interests regarding this manuscript.

**Ethics approval** The ethics approval for the form was obtained from Massey University Ethics Committee. More information can be given if requested.

**Consent for publication** Not applicable



## References

Adam, I. O., Effah, J., & Boateng, R. (2019). Activity theory analysis of the virtualisation of teaching and teaching environment in a developing country university. *Education and Information Technologies, 24*(1), 251–276. https://doi.org/10.1007/s10639-018-9774-7

Amelung, M., Krieger, K., & Rösner, D. (2011). E-Assessment as a service. *IEEE Transactions on Learning Technologies, 4*(2), 162–174. https://doi.org/10.1109/TLT.2010.24

Barra, E., López-Pernas, S., Alonso, Á., Sánchez-Rada, J. F., Gordillo, A., & Quemada, J. (2020). Automated assessment in programming courses: A case study during the COVID-19 Era. *Sustainability*. https://doi.org/10.3390/su12187451

Basharina, O. K. (2007). An activity theory perspective on student-reported contradictions in international telecollaboration. *Language Learning & Technology, 11*(2), 82–103.

Belhaoues, T., Bensebaa, T., Abdessemed, M., & Bey, A. (2016). AlgoSkills: an ontology of Algorithmic Skills for exercises description and organization. *Journal of e-Learning and Knowledge Society, 12*(1), 1826–6223.

Bey, A., Jermann, P., & Dillenbourg, P. (2018). A comparison between two automatic assessment approaches for programming an empirical study on MOOCs. *Journal of Educational Technology & Society, 21*(2), 259–272.

Computing_Curricula_2020_Task_Force. (2021). *Computing Curricula Report 2020* (ISBN: 978–1–4503–9059–0 ). Retrieved from New York: https://dl.acm.org/citation.cfm?id=3467967

Daniels, H. (2004). Activity theory, discourse and Bernstein. *Educational Review, 56*(2), 121–132. https://doi.org/10.1080/0031910410001693218

Daradoumis, T., MarquèsPuig, J. M., Arguedas, M., & CalvetLiñan, L. (2019). Analyzing students' perceptions to improve the design of an automated assessment tool in online distributed programming. *Computers & Education, 128*, 159–170. https://doi.org/10.1016/j.compedu.2018.09.021

Davis, M. (1958). *Computability & unsolvability*. McGraw-Hill.

Engeström, Y. (1999). *Perspectives on activity theory* (pp. 19–38). Cambridge University Press.

Gárcia-Mateos, G., & Fernández-Alemán, J. L. (2009). *A course on algorithms and data structures using on-line judging*. Paper presented at the Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education, Paris, France. https://doi.org/10.1145/1562877.1562897

Gordillo, A. (2019). Effect of an instructor-centered tool for automatic assessment of programming assignments on students' perceptions and performance. *Sustainability*. https://doi.org/10.3390/su11205568

Insa, D., & Silva, J. (2018). Automatic assessment of Java code. *Computer Languages, Systems & Structures, 53*, 59–72. https://doi.org/10.1016/j.cl.2018.01.004

Lemay, D. J., Basnet, R. B., Doleck, T., Bazelais, P., & Saxena, A. (2021). Instructional interventions for computational thinking: Examining the link between computational thinking and academic performance. *Computers and Education Open*, *2*, 100056. https://doi.org/10.1016/j.caeo.2021.100056

Luxton-Reilly, A., Simon, Albluwi, I., Becker, B. A., Giannakos, M., Kumar, A. N., Ott, L., Paterson, J., Scott, M. J., Sheard, J., & Szabo, C. (2018). *Introductory programming: a systematic literature review. Paper presented at the Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, Larnaca, Cyprus*. https://doi.org/10.1145/3293881.3295779

Manzoor, H., Naik, A., Shaffer, C. A., North, C., & Edwards, S. H. (2020). *Auto-Grading Jupyter Notebooks. Paper presented at the Proceedings of the 51st ACM Technical Symposium on Computer Science Education, Portland, OR, USA*. https://doi.org/10.1145/3328778.3366947

Mathrani, S., Mathrani, A., & Khatun, M. (2020). Exogenous and endogenous knowledge structures in dual-mode course deliveries. *Computers and Education Open, 1*, 100018. https://doi.org/10.1016/j.caeo.2020.100018

Medeiros, R. P., Ramalho, G. L., & Falcão, T. P. (2019). A Systematic literature review on teaching and learning introductory programming in higher education. *IEEE Transactions on Education, 62*(2), 77–90. https://doi.org/10.1109/TE.2018.2864133

Murphy, E., & Rodriguez-Manzanares, M. A. (2008). Using activity theory and its principle of contradictions to guide research in educational technology. *Australasian Journal of Educational Technology*. https://doi.org/10.14742/ajet.1203

Park, Y., & Jo, I.-H. (2017). Using log variables in a learning management system to evaluate learning activity using the lens of activity theory. *Assessment & Evaluation in Higher Education, 42*(4), 531–547. https://doi.org/10.1080/02602938.2016.1158236

Parsons, D., Susnjak, T., & Mathrani, A. (2016). Design from detail: Analyzing data from a global day of coderetreat. *Information and Software Technology, 75*, 39–55. https://doi.org/10.1016/j.infsof.2016.03.005

Restrepo-Calle, F., RamírezEcheverry, J. J., & González, F. A. (2019). Continuous assessment in a computer programming course supported by a software tool. *Computer Applications in Engineering Education, 27*(1), 80–89. https://doi.org/10.1002/cae.22058

Richey, R. C., Klein, J. D., & Nelson, W. A. (2004). Developmental Research: Studies of Instructional Design and Development. *Handbook of research on educational communications and technology* (2nd ed., pp. 1099–1130). Lawrence Erlbaum Associates Publishers.

Rubio-Sánchez, M., Kinnunen, P., Pareja-Flores, C., & Velázquez-Iturbide, J. Á. (2012, 29–31 Oct. 2012). *Lessons learned from using the automated assessment tool "Mooshak". Paper presented at the 2012 International Symposium on Computers in Education (SIIE), Andorra la Vella, Andorra*.

Skalka, J., & Drlik, M. (2020). Automated Assessment and Microlearning Units as Predictors of At-Risk Students and Students' Outcomes in the Introductory Programming Courses. *Applied Sciences*. https://doi.org/10.3390/app10134566

Soll, M., Johannsen, M., & Biemann, C. (2021). *Enhancing a Theory-Focused Course Through the Introduction of Automatically Assessed Programming Exercises – Lessons Learned*. Universit¨at Hamburg. Hamburg, Germany. Retrieved from http://ceur-ws.org/Vol-2676/paper6.pdf

Souza, D. M. d., Felizardo, K. R., & Barbosa, E. F. (2016). *A Systematic Literature Review of Assessment Tools for Programming Assignments. 2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)* (pp. 147–156).

Stallman, R. M., & GCC DeveloperCommunity. (2009). *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Scotts Valley, CA, 2009. ISBN 144141276X.

Staubitz, T., Klement, H., Renz, J., Teusner, R., & Meinel, C. (2015, 10–12 Dec. 2015). *Towards practical programming exercises and automated assessment in Massive Open Online Courses. Paper presented at the 2015 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*.

Su, S., Zhang, E., Denny, P., & Giacaman, N. (2021). *A Game-Based Approach for Teaching Algorithms and Data Structures using Visualizations. Paper presented at the Proceedings of the 52nd ACM Technical Symposium on Computer Science Education, Virtual Event, USA*. https://doi.org/10.1145/3408877.3432520

Tong, A., Sainsbury, P., & Craig, J. (2007). Consolidated criteria for reporting qualitative research (COREQ): A 32-item checklist for interviews and focus groups. *International Journal for Quality in Health Care, 19*(6), 349–357. https://doi.org/10.1093/intqhc/mzm042

Twining, P., Heller, R. S., Nussbaum, M., & Tsai, C.-C. (2017). Some guidance on conducting and reporting qualitative studies. *Computers & Education, 106*, A1–A9. https://doi.org/10.1016/j.compedu.2016.12.002

Ullah, Z., Lajis, A., Jamjoom, M., Altalhi, A., Al-Ghamdi, A., & Saleem, F. (2018). The effect of automatic assessment on novice programming: Strengths and limitations of existing systems. *Computer Applications in Engineering Education, 26*(6), 2328–2341. https://doi.org/10.1002/cae.21974

Watson, C., & Li, F. W. B. (2014). *Failure rates in introductory programming revisited. Paper presented at the Proceedings of the 2014 conference on Innovation & technology in computer science education, Uppsala, Sweden*. https://doi.org/10.1145/2591708.2591749

Zvacek, S. M. (2015). *From know-how to know-why: Lab-created learning. Paper presented at the 2015 3rd Experiment International Conference (exp.at'15), Ponta Delgada, Portugal*. https://doi.org/10.1109/EXPAT.2015.7463260



**Andre L. C. Barczak** received the B.Eng. degree in mechanical engineering from Unicamp, Brazil, in 1987, the M.S. degree from Unicamp, Brazil, in 1996 and the Ph.D. degree in computer science from Massey University, New Zealand, in 2008. From 1989 to 1995, he was working as a system engineer with IBM. He was an assistant researcher with Unicamp from 1995 to 1998. In 1998 he emigrated to New Zealand and worked as a system engineer until 2002. From 2002 to 2022, he has worked as an academic staff with Massey University in computer science. Since 2023 he is with Bond University in Australia. He has published more than 70 technical articles in journals and conferences. His research interests include computer vision, machine learning, parallel systems, big data and fundamental algorithms.

**Anuradha Mathrani** received the B.Tech. degree in electronics engineering from Allahabad University, India, in 1989, the M.M.S. degree in information systems from the University of Pune, India, in 1992, and the Ph.D. degree in information technology from Massey University, New Zealand, in 2009. From 1990 to 1995, she was a Researcher with the Philips Electronics Research and Development Laboratory. She was a Lecturer with Allahabad University, from 1996 to 1998, and Pune University, from 1999 to 2001. Since 2002, she has been an Academician in information technology with Massey University. She has published more than 80 articles in highly ranked journals and conferences. Her research interests include technology

enhanced education, computer simulation, and software quality practices. She is currently a Senior Fellow of the Higher Education Academy.

**Binglan Han** received the B.Eng. degree in electronics and information systems from Inner Mongolia University, China, in 1989, the M.Sc. degree in computer science from Massey University, New Zealand, in 2001. From 1989 to 1997, she was working as a computer and electronics system engineer with Air China. She was a senior tutor with Massey University from 2002 to 2014, a lecturer with Auckland Institute of Studies from 2015 to 2016, and an online educator with the Mind Lab from 2016 to 2019. Since 2019, she has been a senior tutor in computer science and information technology with Massey University. She has published several articles in conferences and journals. Her research interests include learning analytics and web-based educational systems.

**Napoleon H. Reyes** received a BSc. degree in Physics, in 1993, at De La Salle University (DLSU), Philippines. As a young physicist who was keen to apply his knowledge in the industry, he immediately joined a software development company named, Vision-8, as a product design engineer. On a contractual project at Intel Philippines, he was exposed to the special needs of the industry and became more interested in programming and algorithms. It wasn't long before he realised that he wanted to pursue a postgraduate degree in computer science, which also lead him to pursue a career in the academe. In 2004, he received a Ph.D. in computer science with high distinction award at DLSU, Philippines. The following year, he emigrated to New Zealand, to work for Massey University. He has published more than 50 conference papers and journal articles, mostly on the topic of intelligent systems, machine learning and computer vision